VI.07.

Технические вопросы, связанные с интерфейсами, почти все мы рассмотрели. Какое-то понимание интерфейсов и полиморфзма, кажется, есть. И сегодня начнём разговор о технологии - о применении интерфейсов, об особенностях их использования именно в Go-шной модели интерфейсов, о специфике решения вопросов реализации полиморфизма в Go. Связаны эти особенности в первую очередь с тем, что интерфейсы в Go реализуются неявно. Но обо всём по порядку.

И снова обратимся к статье *Martin Kock. A Guide to Interfaces in Go*, расположенной здесь : https://fast4ward.online/posts/a-guide-to-interfaces-in-go/, которая активно использовалась на прошлом занятии. Рассмотрим пример из этой статьи:

Использование интерфейсов

У нас имеется некоторая база данных, что бы это ни значило. Может быть это база данных с доступом через SQL, может быть это просто какой-то файл или набор взаимосвязанных файлов, в общем, форма хранения несущественна. Рассмотрим интерфейс EntityLoader, который загружает различные объекты из базы данных по некоторому целочисленному идентификатору. И для того, чтобы абстрагироваться от различий в реализациях хранения данных в базе, воспользуемся как раз интерфейсом. Для каждого вида объектов используем свою реализацию интерфейса.

В первом приближении решение будет выглядеть так:

```
type EntityLoader interface {
    Load(id uint32) (interface{}, error)
}
```

Да, возвращаем мы пустой интерфейс - сущности в базе данных могут храниться самые различные. Да даже если и не совсем различные, если мы можем сказать, что сущности в базе имеют что-то общее, и, тем самым функция `Load` возвращает некоторый именованный интерфейс, ничего по сути не меняется. Но пусть будет пустой интерфейс.

Это работает не слишком хорошо - нам придется вручную преобразовать возвращаемый интерфейс в правильную сущность, будь то User, Address, PurchaseOrder или что-то другое. Чтобы получить точные типы всего, что мы загружаем, можно было бы сделать что-то вроде:

```
type EntityLoader interface {
    LoadUser(id uint32) (User, error)
    LoadAddress(id uint32) (Address, error)
    LoadPurchaseOrder(id uint32) (PurchaseOrder, error)
}
```

Каждый из методов возвращает сущность соответствующего типа и nil-ошибку, если сущность с заданным идентификатором является объектом именно этого типа. Иначе возвращается непустая ошибка.

И зачем нам тогда интерфейсы? Более того, для каждого репозитория, для каждого способа хранения нам потребуется реализовывать все три метода.

В чём же корень проблемы? Где-то мы вступаем в противоречие с самой сутью понятия интерфейса…

Дело здесь в том, что приемник метода не является частью интерфейса, он находится снаружи интерфейса, не входит в контракт интерфейса. Соответственно, есть смысл обрабатывать изменяющуюся часть возвращаемого интерфейса именно в приёмнике, а не в репозитории. Для достижения этой цели метод репозитория может возвращать значение некоторого именованного интерфейса, который может быть реализован любым типом хранящихся данных, тех данных, которые мы загружаем.

Давайте попробуем. Мы назовем новый интерфейс Filler, в нём есть метод ReadFrom(row), который вычитывает, выцарапывает сущность из строки базы. Чуть подробнее: аргументом интерфейса является строка базы данных, а его реализация состоит в том, чтобы заполнить поля сущности базового/конкретного типа соответствующими столбцами из этой строки:

```
type EntityLoader interface {
    Load(id uint32, h Filler) error
}

type Filler interface {
```

```
    ReadFrom(databaseRow RowType) error
}
```

Посмотрим на реализацию этого плана для сущности User:

```
type UserRepository ...

func (u *UserRepository) Load(id uint32, h Filler) error {
    row := ... // row получает значение строки репозитория, соответствующей ид
ентификатору id
    return h.ReadFrom(row)  // h - это указатель на сущность, на величину базо
вого типа, реализующего интерфейс Filler.
    // Возвращаемая ошибка - это возможная ошибка,
    // например ошибка при чтении репозитория или строка с
    // идентификатором id содержит сущность, не соответствующую
    // базовому типу переданного интерфейса Filler.
    // Если err == nil, то в результате h* заполнена информацией, содержащейся
 в row.
}
```

и

```
type User struct {
    ID uint32
    Name string
    // Other fields...
}

func (r *User) ReadFrom(databaseRow RowType) (err error) {
    // Проверяет, соответствует ли содержание databaseRow хранению величны тип
а User.
    // Зачастую это может исполняться прямо методом Unmarshal типа RowType,
    // например, gob.Encode или json.Unmarshal устроены именно так - они
    // сами видят куда они закачивают результат и при несоответствии выкатываю
т ошибку.
    // Но в общем случае проверка соответствия строки ожидаемой сущности -
    // соответствия строки databaseRow типу ресивера r* - есть
    // функция Filler'а (т.е. метода ReadFrom), а не метода Unmarshal.
    // При несоответствии устанавливает err.
    err = databaseRow.Unmarshal(r)
```

```
    ir err != nil {
        return
    }
    // Метод Unmarshall заполняет поля r* соответствующими значениями,
    // возможно проверяя при этом, хранит ли строка databaseRow запись,
    // соответствующую сущности User. Если проверка производится, то в
    // случае удачного исхода возвращает nil, иначе - соответствующую ошибку
    return
}
```

Таким образом, именно метод ReadFrom (свой для каждого типа сущностей, хранящихся в базе данных, поскольку все они реализуют интерфейс Filler) и содержит анализ строки базы данных, именно он и знает и распознаёт все типы сущностей, которые хранятся в базе, возможно, привлекая для этого метод Unmarshal строки базы данных. Всё срастается.

Вот и все, мы можем использовать его вот так:

```
    var u User
    repo := new(UserRepository)
    err := repo.Load(42, &u)
    if err == nil {
        ...
    } else {
        игнорируем строку или
        пытаемся определить тип сущности, хранящейся в строке базы с идентифик
атором 42
    }
```

Избегая, таким образом, вот такого:

```
    maybeUser, err := repo.LoadUser(42) // Returns interface{}, error
    if err != nil { ... }
    user, ok := maybeUser.(*User) // Ugly conversion
    if !ok { ... }
```

Обратите внимание, как мы перевернули отношения между UserRepository и User. Вместо того, чтобы полагаться на репозиторий для заполнения User, мы расширяем возможности User’а для заполнения на основе какого-либо общего параметра.

Переворачивание этих отношений и позволяет нам справиться с той частью методов интерфейса, которая варьируется в разных реализациях.

В Go полиморфизм реализуется, концентрируясь, на том, что можно делать с переменной.

В Go интерфейсы реализуют полиморфизм, позволяя игнорировать тип переменной, а вместо этого сконцентрироваться на том, что мы можем ДЕЛАТЬ с ней. Не что она может делать, а что мы можем с ней делать. И это очень важный момент, и с ним мы будем разбираться минимум ещё несколько занятий. В чём существенность? Объектная модель подразумевает “разумность” данных - данные просто по факту появления уже обладают некоторыми реализованными возможностями, а мы более или менее тупо (или изысканно, как угодно можно это назвать) призываем данные использовать эти свои навыки и умения. Как только в поле зрения появляется полиморфизм в любом виде, ключевым вопросом становится вопрос об RTTI - Run-Time Type Information, о распознавании конкретного типа величины в процессе выполнения кода. И, разумеется, тут появляется широкое поле для всевозможных несоответствий типов и связанных с этим проблем. Грубо говоря, никто и никогда не пишет код целиком и “с нуля”, пользуются какими-то библиотеками, пакетами, готовыми функциями, ещё какими-то другими структурными единицами кода. И даже не потому, что библиотеки - это хорошо, а потому, что надо же код как-то организовывать, это же немыслимо хранить код тупо в одном файле. И в одной голове - и это таки важно. И вполне возможно, что где-то, причём достаточно далеко от нашего личного кода, кода, который мы вот сейчас пишем, предположения о свойствах передаваемой величины не будут соответствовать нашим предположениям. Этот момент - это как раз чуть ли не самый опасный момент во всей этой истории. И бороться за безопасность кода по типам (type safety - безопасность типов, если дословно) - это очень важно. Объектная модель в более или менее чистом виде направлена на делегирование алгоритмов обработки данных типам, тем самым унося, уводя их от пользователя (пользователя типов, разумеется, от программиста, а не от конечного пользователя). В Go мы концентрируемся на собственных усилиях, мы сами расширяем возможности типа для осуществления более или менее общей обработки, обработки общего вида на основании каких-то общих параметров интерфейса.